Analyzing and Leveraging Shared L1 Caches in GPUs

Mohamed Assem Ibrahim
William & Mary
maibrahim@email.wm.edu

Onur Kayiran
Advanced Micro Devices, Inc.
onur.kayiran@amd.com

Yasuko Eckert
Advanced Micro Devices, Inc.
yasuko.eckert@amd.com

Gabriel H. Loh
Advanced Micro Devices, Inc.
gabriel.loh@amd.com

Adwait Jog
William & Mary
ajog@wm.edu

ABSTRACT

Graphics Processing Units (GPUs) concurrently execute thousands of threads, which makes them effective for achieving high throughput for a wide range of applications. However, the memory wall often limits peak throughput. GPUs use caches to address this limitation, and hence several prior works have focused on improving cache hit rates, which in turn can improve throughput for memory-intensive applications. However, almost all of the prior works assume a conventional cache hierarchy where each GPU core has a *private* local L1 cache and all cores share the L2 cache. Our analysis shows that this canonical organization does not allow optimal utilization of caches because the private nature of L1 caches allows multiple copies of the same cache line to get replicated across cores.

We introduce a new *shared* L1 cache organization, where all cores collectively cache a single copy of the data at only one location (core), leading to zero data replication. We achieve this by allowing each core to cache only a non-overlapping slice of the entire address range. Such a design is useful for significantly improving the collective L1 hit rates but incurs latency overheads from additional communications when a core requests data not allowed to be present in its own cache. While many workloads can tolerate this additional latency, several workloads show performance sensitivities. Therefore, we develop lightweight communication optimization techniques and a run-time mechanism that considers the latency-tolerance characteristics of applications to decide which applications should execute in private versus shared L1 cache organization and reconfigures the caches accordingly. In effect, we achieve significant performance and energy efficiency improvements, at a modest hardware cost, for applications that prefer the shared organization, with little to no impact on other applications.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

Bandwidth, GPUs, Locality

PACT '20, October 3–7, 2020, Virtual Event, GA, USA

ACM ISBN 978-1-4503-8075-1/20/10…$15.00
https://doi.org/10.1145/3410463.3414623

ACM Reference Format:
Mohamed Assem Ibrahim, Onur Kayiran, Yasuko Eckert, Gabriel H. Loh, and Adwait Jog. 2020. Analyzing and Leveraging Shared L1 Caches in GPUs. In *Proceedings of the 2020 International Conference on Parallel Architectures and Compilation Techniques (PACT '20), October 3–7, 2020, Virtual Event, GA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3410463.3414623

1 INTRODUCTION

Graphics Processing Units (GPUs) have emerged as very effective general-purpose accelerators for a wide range of applications. They have been successful because they provide very high throughput at a competitive power budget. High-bandwidth memories provide the foundation for supporting the fine-grain multithreading that GPUs rely upon for achieving high throughput. However, the well-known memory wall [68] is often the performance-limiting factor for GPUs. Traditionally, a popular approach to address the memory wall problem has been to employ on-chip memories such as caches. In CPUs, caches have been very effective in cutting down memory latencies. In GPUs, however, latency is not often the first-order challenge for many applications because of the high level of multithreading. Still, GPUs are equipped with both software-managed (scratchpad) and hardware-managed on-chip memories (caches) to reduce traffic to the lower levels of the memory hierarchy. An increase in on-chip memory hit rate can lead to a proportional decrease in memory traffic, translating into performance improvements for memory-intensive programs [45, 67]. Therefore, researchers in the past have invested significant efforts in improving cache performance via hardware and software methods [24, 26, 27, 29, 32, 54, 73].

GPUs typically employ a two-level cache hierarchy, where each core is associated with a private local L1 cache, and all cores in the GPU share a banked L2 cache. An interconnect connects all cores to the L2 caches and memory partitions. The L1 caches are responsible for reducing traffic to the interconnect and L2 cache, while the L2 cache helps to reduce memory traffic. This paper challenges such a conventional cache organization and reveals inefficiencies in the existing cache hierarchy in the context of GPUs. In particular, we focus on addressing the inefficiencies associated with GPUs' *private* local L1 caches. Specifically, because of the private nature of the L1 caches, the same cache lines can be requested by different cores, leading to high inter-core locality [15, 23, 33, 40, 41]. This data (cache line) replication reduces the effective aggregate capacity of the L1 caches across all cores, leading to their lower bandwidth utilization as we will show in Section 2.

To address these challenges, we propose and evaluate *shared* local L1 caches in GPUs. The key idea is to ensure only one copy of data exists across L1 caches, thereby eliminating data replication

and making better use of the finite cache capacity. We propose to realize the shared L1 caches by making minimal changes to the existing L1 cache controller and address mapping policies, with *no* changes to the L1 caches. Normally, each core can cache any data from the entire address range. Instead, our shared L1 cache design restricts each core to cache only a unique slice of the address range. Consequently, each core caches data from non-overlapping address ranges, which eliminates data replication across local caches.

Although such a design is attractive for GPUs, it requires inter-core communication if one core requests data that is not mapped to its allocated address range. In such situations, additional latency will be incurred to fetch the data from the L1 cache of a remote core. Fortunately, thanks to the latency-tolerance of many GPGPU applications, an increase in latency often has a negligible impact on performance. However, not all applications a) can tolerate long memory latencies, b) exhibit data replication, or c) are sensitive to cache capacity (i.e., their working sets fit in L1 cache or they stream with little-to-no locality). Consequently, shared local caches can have negative or no effect on such applications' performance. To address these concerns, we develop lightweight mechanisms to a) reduce the inter-core communication overhead and b) identify applications that prefer the private L1 organization and hence execute them accordingly.

Contributions: This paper contributes the following:

- We propose shared L1 caches in GPUs. To the best of our knowledge, this is the first paper that performs a thorough characterization of shared L1 caches in GPUs and shows that they can significantly improve the collective L1 hit rates and reduce the bandwidth pressure to the lower levels of the memory hierarchy.
- We develop GPU-specific optimizations to reduce inter-core communication overheads. These optimizations are vital for maximizing the benefits of the shared L1 cache organization.
- We develop a GPU-specific lightweight dynamic scheme that classifies application phases and reconfigures the L1 cache organization (shared or private) based on the phase behavior.
- We extensively evaluate our proposal across 28 GPGPU applications. Our dynamic scheme boosts performance by 22% (up to 52%) and energy efficiency by 49% for the applications that exhibit high data replication and cache sensitivity without degrading the performance of the other applications. This is achieved at a modest area overhead of 0.09 mm^2/core.
- We make a case to employ our dynamic scheme for deep-learning applications to boost their performance by 2.3×.

2 MOTIVATION AND ANALYSIS

In this section, we first quantify the data replication problem associated with private L1s in GPUs (as described in Section 1) and then make a case for shared L1s to address this inefficiency.

2.1 Analysis of Wasted L1 Cache Space

Figure 1 shows the cache line replication ratio under the baseline private L1 organization for the evaluated applications (methodology detailed in Section 5.1). The cache line replication ratio is defined as the ratio of L1 misses that can be found in other L1 caches to total L1 misses. We observe that the replication ratio varies across the applications. Specifically, some applications have no replication

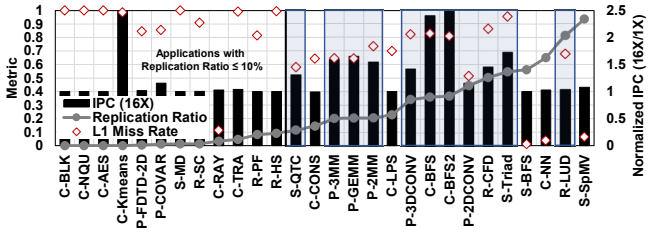


Figure 1: Performance of the evaluated applications in terms of L1 miss rate, cache line replication ratio, and IPC improvement under 16× the L1 cache size (normalized to baseline). The left-hand y-axis represents cache line replication ratio and raw L1 miss rate.

(e.g., C-BLK) or low replication (e.g., C-RAY), while others have high replication (e.g., C-BFS).

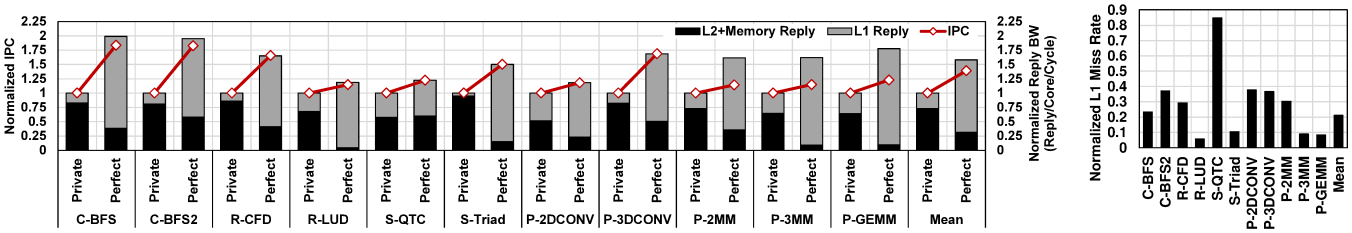
Identifying Target Applications. The waste due to data replication may not affect all applications. Only the applications that are sensitive to larger cache space are *expected* to benefit if the wasted cache space is reduced/eliminated. Therefore, we study their performance under a 16× larger L1 cache in Figure 1. We observe that 13 applications are both capacity-sensitive and possess high data replication. To identify the subset of the capacity-sensitive applications that are sensitive to data replication, we study their L1 miss rates. Applications with low L1 miss rates (e.g., C-NN and S-SpMV) may not suffer under private L1 caches because the majority of their requests can be satisfied locally. These applications tend to have working sets smaller than the baseline L1 cache capacity. In general, we consider an application to be sensitive to data replication if it 1) has a replication ratio of >10%, 2) has an L1 miss rate of >50%, and 3) observes a speedup of >5% with 16× capacity.[1] Based on these criteria, we observe that 11 applications are sensitive to data replication (marked by the blue boxes in Figure 1). These are our target applications.

2.2 A Case for Shared L1 Caches

One way to eliminate data replication is to enable a shared cache organization across the local L1 caches. Under a private L1 organization, each core can cache any line. For example, given four different address ranges represented by different shades in Figure 3a, a private L1 cache can store any cache line from all four address ranges. However, under a shared L1 organization, the entire address range is interleaved across all cores and such mapping is fixed. In other words, each core caches data from a non-overlapping address range. For example, as shown in Figure 3b, the address range represented by white can be cached by only L1-0, and the address range represented by black can be cached by only L1-3. Because an exclusive slice of the address range maps to a single L1, the shared L1 organization ensures no cache line replication across L1s. However, to fully unlock the potential of the shared L1 organization, the cores need to communicate to fetch the data that do not belong to their assigned address ranges.

Sources of Benefits. To understand the scope of potential performance benefits of the shared L1 organization, we set up a hypothetical design where all cores can communicate with each other

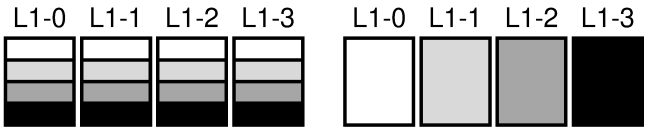
[1]This criteria is empirical and is not used by our proposed scheme in Section 4.



(a) IPC and reply bandwidth

(b) L1 miss rate

Figure 2: Performance of a hypothetical cache design that eliminates data replication across local L1 caches. Results are normalized to the private L1 baseline. Section 5.1 has the details on the experimental methodology.



(a) Private per-core L1 caches. All L1s can cache any address range.

(b) Shared L1 caches across cores. Each L1 can cache only an exclusive address range.

Figure 3: Private and shared cache organizations.

with zero cycle overhead and share their L1 caches ensuring no data replication. Figure 2 shows performance in terms of IPC and L1 miss rate for the identified target applications executing on this hypothetical system, normalized to the private L1 baseline. We observe, in Figure 2a, that such a hypothetical configuration improves performance by between 14% and 83% across these applications, and 39% on average. The main reason for such a performance boost is the significant 79% reduction in the collective L1 miss rate (shown in Figure 2b) that results in better L1 bandwidth utilization (i.e., total collective *useful* bandwidth received from the L1 hits is higher than the baseline).[2] Consequently, the L2 and memory bandwidth consumption is reduced, thereby making the L1s more effective at addressing the memory wall challenge.

Overall, we conclude that the shared L1 organization eliminates wasted L1 cache capacity and thus can lead to a significant performance improvement for the target applications. Therefore, we refer to them as *shared-friendly* applications. Next, we propose a shared L1 cache design that aims to achieve the performance of this hypothetical cache design for the shared-friendly applications.

3 SHARED L1 CACHES: DESIGN, ANALYSIS, AND OPTIMIZATIONS

In this section, we describe our design that enables both private and shared L1 organizations, and demonstrate the potential performance of a realistic shared L1 organization.

3.1 Terminology and Address Mapping

Under a shared L1 organization, we define two terms that we use in this paper: *requester* and *home* cores. A requester is a core that requests a given cache line and the home is the core that can cache that line. For example, in Figure 3b, the home core of a line that falls in the black address range is core $L1-3$. If core $L1-3$ requests that line, then the core is both the requester and the home for that line. Additionally, a typical memory access under a shared cache organization can be either *local* or *remote*. An access is considered local if the requester core is also the home core. Otherwise, an access is remote. For example, in Figure 3b, if core $L1-0$ requests a cache line from the black address range, then it will send a remote request to the home core $L1-3$.

Selecting the Home Core. To select the home core for a given cache line, we use *core bits*. These core bits are selected from the physical address of a request. The process of selecting these bits is analogous to selecting the DRAM bank bits based on the physical address. In the private L1 cache organization, there are no core bits because the requester is always the home. In a system with N local L1 caches (each attached to one core) that are organized in a shared fashion, we use the least significant $\lceil log_2(N) \rceil$ bits of the tag as core bits to select the home core for a given cache line. Because the core count or the cache being organized as private or shared does not affect the number of tag bits according to this mapping policy, our system always uses 20 bits as tag.

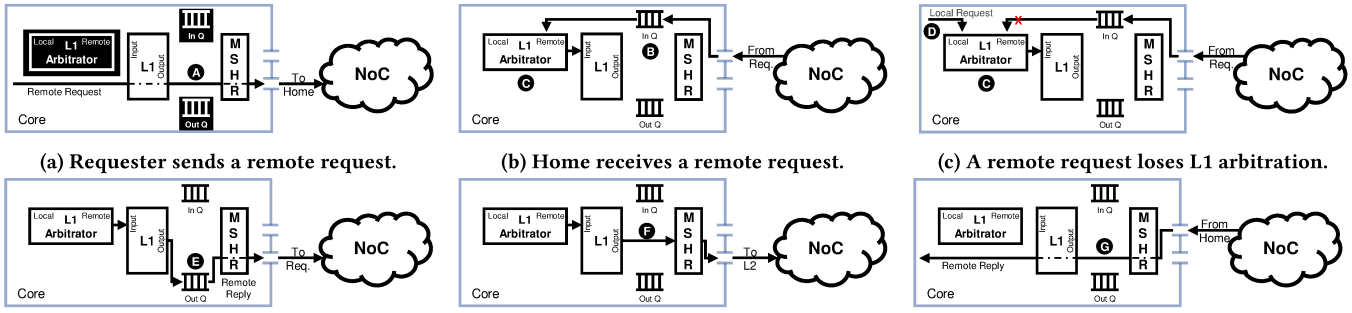
3.2 Shared L1 Caches Design

Figure 4 shows the communication flow in a simple design that enables the L1 caches to be organized as either private or shared. Each core is connected to an L1 cache, which has associated Miss Status Holding Registers (MSHRs) to track pending L1 misses. The MSHRs are connected to the network-on-chip (NoC) that routes L1 misses to the L2. In the baseline private L1 organization, each core sends requests to its local L1, and the misses go through its local MSHR to access the L2 via the NoC.

Handling Read Requests. With a shared L1 organization, a remote read request skips the L1 cache of the requester core because the data cannot be there. It then also skips the local MSHR Ⓐ and goes through the NoC to reach the home core. The home core queues the received remote request Ⓑ and consults its local L1 cache arbitrator Ⓒ, which prioritizes the local cache requests over remote requests. If there are no local requests, the remote request accesses the L1 cache of the home core. Otherwise, the local request is processed Ⓓ and the remote request remains queued. If the request hits in the home L1 cache, then the L1 queues the read reply Ⓔ for injection into the NoC back to the requester. If the request misses in the home L1 cache, then the home core sends the request to the L2 cache through the NoC Ⓕ.[3] Once a read reply is received from the L2 via the NoC, the home core installs the reply in its

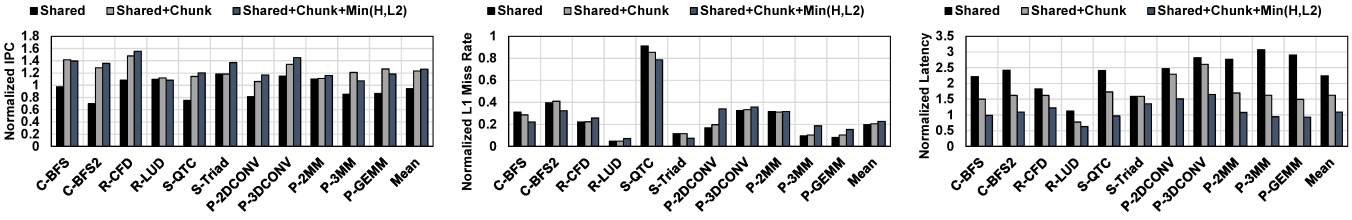
[2] L1 and L2+Memory reply bandwidth represent the number of replies received from L1 and L2+Memory, respectively, over the total execution time.

[3] A single MSHR entry is allocated for a unique cache line address at the home's L1 to allow coalescing of misses originating from both local and remote read requests.



(a) Requester sends a remote request.
(b) Home receives a remote request.
(c) A remote request loses L1 arbitration.
(d) Home sends a remote reply.
(e) Home handles a remote request miss.
(f) Requester receives a remote reply.

Figure 4: Request/Reply flow in a shared L1 organization. The L1 Arbitrator and the In/Out queues, shown in black in (a), are newly added to support our proposal. Dashed lines represent L1/MSHR bypassing.



(a) IPC
(b) L1 miss rate
(c) Average latency on the reply network.

Figure 5: Performance of a realistic shared L1 organization. Results are normalized to the private L1 baseline.

local L1 and concurrently queues the reply Ⓑ to be injected to the requester through the NoC. Finally, the requester core receives and processes the remote read reply without caching it locally Ⓖ.

Handling Write Requests. With a shared L1 organization, a remote write request follows the same flow as a remote read request. However, a remote write request always skips the MSHR of both the requester and the home. Also, we use write-through and no-write-allocate policies in the L1 caches (Section 5.1). Therefore, on a write hit, a given write request modifies the cache line in the home core. The modified cache line is forwarded to the L2 cache through the NoC. However, on a write miss, no cache line is allocated at the home core and the updated data is delivered to the L2 cache. Once a write ACK is received from the L2, the home core forwards the write ACK to the requester core via the NoC.

Handling Coherence. With a shared L1 organization, only a single copy of a cache line may exist across L1s. Therefore, there may not be a need for coherence mechanisms within a single GPU.

Handling Non-L1 Requests All non-L1 (instruction, texture, and constant cache) misses from the GPU core are not affected by the shared cache organization. Non-L1 misses are simply forwarded to the L2 via the NoC as in the private L1 baseline.

Handling Atomic Operations. In the baseline, atomic operations skip the L1 cache and are handled at L2/MC (memory controller) [6]. Similarly, in our design, atomic operations skip the requester and home L1 caches and are handled at the unaltered L2/MC.

Communication Fabric. We evaluate shared L1 caches with a mesh interconnect [5, 23, 30, 49, 70] in Section 3.3 and present a case study of a crossbar-based system in Section 5.5. Other interconnect topologies that allow inter-core communication can be used to unlock the full potential of shared L1 caches, but we leave the study of such topologies for future work.

3.3 Performance Analysis and Optimizations

We analyze the impact of shared L1 cache design on the shared-friendly applications in terms of performance, L1 miss rate, and the reply network latency as shown in Figure 5. We observe that although the shared design (denoted as *Shared*) helps in significantly reducing the L1 miss rate by 80% (as expected per our discussion in Section 2.2), it does not translate into performance improvement over the private L1 baseline. In fact, we observe a performance degradation of 5%. This is because of the overhead incurred (average packet latency of the reply traffic increases by 2.2×) due to the additional communication. Therefore, it is essential to analyze this overhead and propose optimizations to alleviate it.

Optimization I: Reducing Wasted NoC Bandwidth. Because the requester does not install data for remote requests in its own L1, fetching the requested data at a full cache line granularity from the home core wastes NoC bandwidth if only a portion of the line is actually requested by the requester. Figure 6 shows how much data within a line is used by the requester cores for shared-friendly applications. "Access=N" denotes that N bytes out of 128, which is the cache line size, are used by the requester. We observe that many applications do not need the entire cache line data and in fact need only a quarter of it most of the time. We apply this known observation [53] in a different context for reducing interconnect

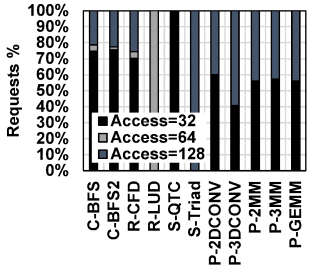


Figure 6: Fraction of useful bytes within a cache line.

traffic between cores. Based on this observation, we design the system such that the data reply from the home to the requester only carries the data requested by the requester, not the entire line. The key idea is to reduce unnecessary data movement and to also avoid wasting precious NoC bandwidth. With the help of this optimization (denoted as *Shared+Chunk*), we observe a significant speedup of 23% for shared-friendly applications as shown in Figure 5a.

Optimization II: Better Distribution of Requests. The next optimization (*Min(H,L2)*) balances the interconnect traffic by selectively routing the L1 requests to either the home L1 or to L2, whichever is fewer hops away. The key idea is to better utilize both the home cores' bandwidth and the L2 bandwidth and cut down latency by going to the nearest source of data. In Figure 5, *Shared+Chunk+Min(H,L2)* shows the effect of applying Min(H,L2) on top of Shared+Chunk. In this experiment, we apply Optimization I (*chunking*) on the traffic from either the home core or L2 to the requester core. We make several key observations. First, with Shared+Chunk+Min(H,L2), we improve the performance benefits to 26%. This is because of the better distribution of interconnect traffic and reduced latency. In Shared+Chunk, all requests go to home cores, which has the potential to create network hotspots and limit the achieved bandwidth from the home cores. With Shared+Chunk+Min(H,L2), there is a better balance between requesters obtaining their data from home cores and L2. Second, Shared+Chunk+Min(H,L2) does not provide significant L1 hit rate benefits, compared to Shared+Chunk. Its performance benefit is mainly because of a more uniform distribution of traffic on the chip, not due to reduced cache contention at the home caches. Finally, Shared+Chunk+Min(H,L2) reduces the latency overhead to 9%, mainly because of lower hop counts and more uniform traffic distribution. We conclude that, for shared-friendly applications, our optimizations can reduce the wasted bandwidth, provide a good balance between miss rate reduction and network latency, and show promising performance improvements. For the rest of the paper, we will refer to Shared+Chunk+Min(H,L2) as *Shared++*.

Evaluating Non-shared-friendly Applications. So far, we have proposed an optimized shared L1 organization and validated its usefulness on the shared-friendly applications. For completeness, we evaluate Shared++ further on other applications (17) that are not classified as shared-friendly (denoted as *non-shared-friendly* applications). Figure 7 shows the performance of these applications normalized to the private L1 baseline. Three observations are in order. First, most of these applications perform as well as the private L1 baseline and are hence classified as *insensitive*. These applications are likely to have a high tolerance to the latency overhead induced by the shared L1 organization. Second, two of these applications (`C-Kmeans` and `P-COVAR`) perform better than the private L1 organization. `C-Kmeans` achieves a 14% performance improvement because of the Min(H,L2) optimization. `C-Kmeans` has high sensitivity to cache size and no replication across cores. Thus, by bypassing the home core and directly going to L2, we effectively increase the cache capacity (increase the L1 hit rate). As for `P-COVAR`, its 20% improvement is because of the work imbalance between the cores in some kernels under the private L1 baseline. Specifically, some kernels do not have enough cooperative thread arrays (CTAs) for all the cores, which leaves the L1 caches of some cores not utilized in the baseline. However, with Shared++, all the L1 caches serve

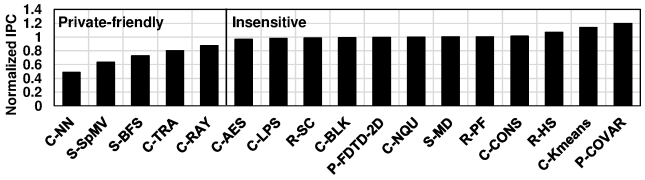


Figure 7: Non-shared-friendly applications under Shared++. Results are normalized to the private L1 baseline.

the requests based on the required address range. Finally, five applications suffer a drop in performance under the proposed shared L1 organization (minimum = 12%, maximum = 51%). We observe that these applications either have high L1 cache locality leading to low L1 miss rates (< 10%) or low latency tolerance. To make a strong case for the shared L1 organization, we need a mechanism that identifies such *private-friendly* applications and executes them in a private L1 organization.

4 A DYNAMIC MECHANISM FOR HANDLING PRIVATE-FRIENDLY APPLICATIONS

In this section, we present a per-core lightweight dynamic scheme that *locally* classifies an application at runtime as shared-friendly or private-friendly and executes the application on a shared or private L1 organization accordingly. Our dynamic scheme utilizes a two-step process: a sampling phase followed by an execution phase. During the sampling phase of a core, it simultaneously collects runtime metrics for both shared and private organizations. Once the sampling phase of a core ends, it evaluates the locally collected information and chooses the desired L1 organization during the next execution phase. After concluding an execution phase, a new sampling phase starts. By repeating this two-step process, our scheme can adapt to the changing behavior of the application.

4.1 Sampling Methodology

In this section, we discuss the details of the sampling mechanism and the per-core collected information as shown in Figure 8.

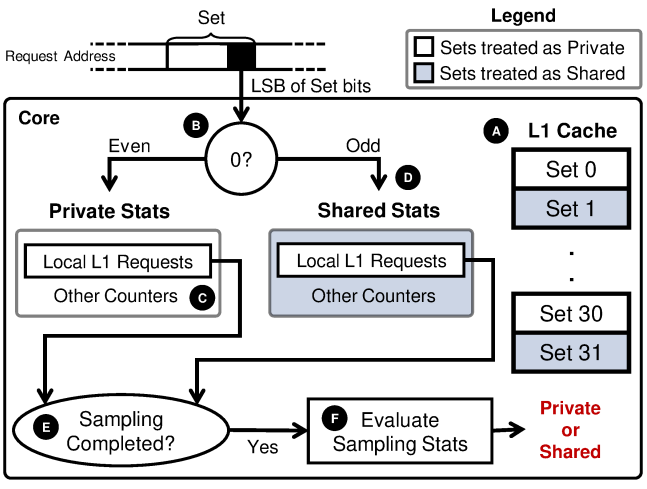


Figure 8: Sampling phase of the dynamic scheme.

Concurrent Evaluation of Private and Shared L1 Organization. Our scheme concurrently evaluates both a shared and a private L1 organization using the local L1 cache during the sampling phase. We accomplish such simultaneous evaluation by treating half of the L1 cache sets as shared and the other half as private. We assign the even sets and the odd sets to be treated as private and shared, respectively Ⓐ. We interleave the set indexing between private and shared at a fine granularity to decrease the bias of requests focusing on a subset of the cache sets. Note that this approach is not a dynamic cache partitioning scheme, thus we do not have the associated overheads [57]. We do not change the indexing of the cache as the set bits are the same. We use the least significant bit (LSB) of the set bits to determine if the required set is even (to be treated as private) or odd (to be treated as shared) Ⓑ.

Sampling Phase. During the sampling phase, we use counters to gather information that is crucial for classifying the running application Ⓒ. For example, we count the number of accesses and misses to the local L1 cache to estimate the L1 miss rate at the end of the sampling phase. Because we evaluate both shared and private cache organizations concurrently, we use two groups of counters for each option, and only the corresponding counters are updated based on the LSB of the set bits. For example, if a core receives a read reply from L2 to install in an odd set, then the replies from L2 counter for the sets that are treated as shared is incremented Ⓓ. The sampling phase continues until both the shared and private groups each process at least R_S local L1 accesses ($R_S = 512$ requests) Ⓔ, where a local L1 access occurs when a core generates a request that is destined to its local L1 cache (i.e., $requester = home$). This makes the time interval for the sampling phase variable. Also, this ensures that each group observes enough requests to have a fair evaluation between the two options.

Execution Phase. Once the sampling phase ends, the counters from each group are used to evaluate which cache organization to use Ⓕ. The evaluation is based on the metrics discussed in Section 4.2. After evaluation, the execution phase starts under the desired L1 organization. The next sampling phase starts after processing R_{Ex} local L1 accesses ($R_{EX} = 16384$ requests).[4]

Selecting the Home Core. Due to the self-paced nature of our dynamic scheme, a given core may be in either sampling or execution phase. Additionally, a core locally chooses the preferred L1 organization. Nevertheless, as discussed in Section 3.1, a core under a shared L1 organization (during sampling or execution) still uses the core bits to determine the home core, even if the home core is under private L1 organization or in a sampling phase.

Handling Coherence. The coherence protocol utilized in the private L1 baseline is used in our dynamic mechanism. Specifically, both the private L1 baseline and our dynamic scheme employ flushing-based software coherence [2, 47, 48, 52, 60, 63, 69].[5] This is ensured by the usage of 1) a write-through L1 cache that is invalidated and flushed at every kernel boundary or at synchronization points, and 2) a shared L2 cache that is inherently coherent. Such system-wide flushing of the L1 caches does not differentiate between a core that is under execution phase (private or shared) or sampling phase. In other words, all L1 caches in the system will be invalidated and flushed indiscriminately at kernel boundary or synchronization points to ensure coherence.

Handling Private-to-Shared Transition. In case shared L1 organization is desired for the execution phase, then some leftover cache lines may exist in the cache. A leftover line is a cache line that was cached during sampling in the sets treated as private but does not belong to the assigned address range of the core. However, if a leftover line is requested, then the core will skip its local L1 cache (as $requester \neq home$) and forward the request to the home core. Thus, these leftovers lines are not utilized by the requester core during the execution phase. Additionally, a request destined to a cache set storing a leftover line will always lead to a tag mismatch with the leftover line as the core bits are different. We employ a lazy invalidation scheme instead of migrating the leftover lines or flushing the L1 cache because of its simplicity. However, the cache replacement policy may be updated to consider the leftover lines for victim-selection. These lines can be identified by using either the core bits or by setting an extra 1-bit per cache line during sampling. Such a policy should replace the leftover lines sooner leading to better cache utilization.

4.2 Sampled Metrics

In this section, we assess the effectiveness of two possible metrics that can be used in classifying an application to be either shared-friendly or private-friendly. A good metric should clearly distinguish between shared-friendly and private-friendly applications with minimum overhead in terms of the sampled information.

Metric I: Average Memory Access Time (*AMAT*) is a well-known metric used to analyze memory system performance in the CPU domain. AMAT is a good candidate for evaluation as it covers the cache capacity aspect (via the miss rate) and reports the average overall latency. For our scheme, AMAT is defined as:

$$AMAT = L1_{HitLatency} + \left(\frac{L}{L+R} \times L1_{LocalMissRate} \times L2_{AccessLatency}\right) + \left(\frac{R}{L+R} \times AMAT_{Home}\right) \quad (1)$$

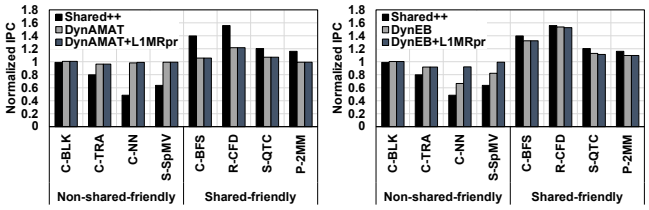
$$AMAT_{Home} = Home_{AccessLatency} + (L1_{RemoteMissRate} \times L2_{AccessLatency}) \quad (2)$$

where L is the number of a core's own local L1 accesses, and R is the number of a core's own remote L1 accesses. $L/(L+R)$ represents a fraction of the given core's own requests that belong to its assigned address range. Similarly, $R/(L+R)$ represents a fraction of the core's own requests that do not belong to its assigned address range.

At the end of the sampling phase, we evaluate AMAT for both shared and private L1 organizations and choose the option with the lower AMAT. Figure 9a shows the effectiveness of AMAT to choose between shared and private L1 organization using four non-shared-friendly applications (one insensitive and three private-friendly) and four shared-friendly applications. We observe that for the non-shared-friendly applications, AMAT (*DynAMAT*) performs as well as the baseline by clearly identifying the insensitive and private-friendly applications. It also performs better than Shared++ for `C-TRA`, `C-NN`, and `S-SpMV`. However, DynAMAT performs poorly

[4] R_S and R_{Ex} values are empirically chosen based on the insight to have longer execution phases to minimize any sampling overheads.

[5] If a hardware-based coherence protocol is used, the directory at L2 will correctly keep track of the list of sharer cores and the invalidations will only be sent to the sharers.



(a) AMAT **(b) EB**

Figure 9: Effect of different metrics on the dynamic scheme.

with the shared-friendly applications, losing the performance benefits gained by using the shared L1 organization. This is because AMAT is oblivious to latency tolerance in GPUs. Thus, even with the latency overhead imposed by the shared L1 organization to access remote home cores, GPUs may be able to hide such an increase in latency due to their huge parallelism. This makes a case for using another metric.

Metric II: Effective Bandwidth (*EB*) is defined as the ratio of bandwidth to miss rate and is calculated based on the level of memory hierarchy under consideration. At a given core, EB is computed as BW/CMR, where $CMR = L1_{MissRate} \times L2_{MissRate}$. EB is a good candidate for the following reasons. First, Wang *et al.* [67] showed that $IPC \propto EB$. Thus by optimizing for a higher EB, we aim for a higher IPC as well. Second, EB is sensitive to the change in the L1 effective capacity as it has an L1 miss rate aspect. Third, EB accounts for latency tolerance in GPUs as well by considering bandwidth. In other words, even if some requests end up incurring high latency, more requests may be processed within the same time interval, increasing the overall received bandwidth. Finally, using EB, we can distinguish the performance impact of requests being cached using a shared or a private organization. However, doing so by using a direct performance metric (e.g., IPC) would be difficult because our scheme deals with requests, not instructions. Furthermore, performance metrics might vary due to reasons other than L1 performance (e.g., bandwidth obtained from software-managed caches [25]), which can lead to an inaccurate classification of applications during runtime. In our scheme, our proxy EB is defined as:

$$EB = \frac{L2_{Replies}}{L1_{MissRate}} + Home_{Replies} \quad (3)$$

where $L2_{Replies}$ and $Home_{Replies}$ are the number of read/write replies from L2 and home core(s), respectively.

At the end of the sampling phase, we evaluate EB for both shared and private L1 organizations and choose the option with higher EB. Figure 9b shows the effectiveness of EB in choosing between shared and private L1 organizations. We observe that EB (*DynEB*) achieves the performance improvement of a shared L1 organization for the shared-friendly applications. As for the non-shared-friendly applications, EB performs as well as private for `C-BLK` and `C-TRA`. However, for `C-NN` and `S-SpMV`, EB falls behind the private L1 organization by up to 33%. To remedy that, we utilize our observation (Section 3.3) that such applications have significantly low L1 miss rates (< 10%) and low latency tolerance.

Optimization. We augment our DynEB by checking if the sets treated as private have an L1 miss rate lower than $L1MR_{Threshold}$ (= 10% in our evaluation). *DynEB+L1MRpr* denotes the updated DynEB in Figure 9b. DynEB+L1MRpr performs as well as the private L1 organization for the non-shared-friendly applications while maintaining the IPC improvement for the shared-friendly applications. We also updated the AMAT-based metric with the L1 miss rate optimization and, as shown in Figure 9a, DynAMAT+L1MRpr is still not effective with the shared-friendly applications.

4.3 Hardware Overhead

As discussed in Section 3.2 and Section 4.1, our optimized shared L1 organization and DynEB do not change the L1 caches or the NoC. We only update the request handling architecture to manage the remote accesses. We synthesized the RTL design of the hardware required for our optimized shared L1 organization using the 65nm TSMC libraries in the Synopsys Design Compiler and estimated the area overhead to be 0.085 mm^2 per core. DynEB leads to an additional area overhead of 0.005 mm^2 per core.

5 EXPERIMENTAL EVALUATION

In this section, we first describe our experimental setup and then evaluate our proposed solutions.

5.1 Experimental Setup

Our baseline architecture assumes a generic GPU, consisting of multiple cores (also called Compute Units, or CUs) that have private local L1 caches. These caches are connected to multiple address-sliced L2 cache banks via a NoC. We use two separate networks: request and reply networks to avoid protocol deadlocks [6]. We faithfully model our shared L1 cache organization, inter-core communication, and other mechanisms using a cycle-level simulator – GPGPU-Sim v.3 [6]. A detailed platform configuration is described in Table 1. We evaluate 28 benchmarks from four suites (CUDA-SDK (C) [46], Rodinia (R) [10], SHOC (S) [13], and PolyBench (P) [50]).

Table 1: Configuration parameters of the simulated GPU.

Core Features	1400MHz core clock, 28 cores (CUs), SIMD width = 32 (16 × 2)
Resources / Core	48KB scratchpad, 32KB register file, Max. 1536 workitems (48 wavefronts, 32 workitems/wavefront)
L1 Caches / Core	16KB 4-way Write-through L1 data cache - Latency = 28 cycles [31] 12KB 24-way texture cache, 8KB 2-way constant cache, 2KB 4-way I-cache, 128B cache block size
L2 Cache	8-way 128 KB/memory channel (1MB in total) 128B cache block size - Latency = 120 cycles
Memory Model	8 GDDR5 Memory Controllers (MCs) FR-FCFS scheduling, 16 DRAM-banks, 4 bank-groups/MC, 924 MHz memory clock, Global linear address space is interleaved among partitions in chunks of 256 bytes [17] Hynix GDDR5 Timing [22]
Interconnect	6 × 6 mesh topology, 700MHz interconnect clock, 32B flit size, 4 VCs per port, 4 flits/VC, iSLIP VC and switch allocators

5.2 Experimental Results

In this section, we evaluate and compare the following against a private L1 organization baseline:

• **Shared++:** Our shared L1 organization augmented with the optimizations in Section 3.3.

• **DynEB:** Our EB-based dynamic scheme, *augmented* with the *L1MRpr* optimization (Section 4.2), to classify applications either as shared-friendly or private-friendly.

• **Best(Private,Shared++):** This configuration statically captures the best of both private and shared L1 organizations by picking the organization that achieves higher IPC.

Effect on Performance. Figure 10 shows the IPC performance of our proposed solutions normalized to the private L1 baseline. We observe the following. First, DynEB exploits the benefits of the shared L1 organization for shared-friendly applications. Specifically, DynEB enhances IPC by 22% on average over the private baseline and is within 3% of Best(Private,Shared++) for the shared-friendly applications. This is because DynEB significantly reduces data replication, thus it increases the effective L1 cache capacity. Second, DynEB compensates for the IPC loss of the private-friendly applications under Shared++. As discussed in Section 3.3, these applications have a significantly low L1 miss rate and high sensitivity to latency. Thus, their performance suffers because, with Shared++, even a cache hit may have to go through the NoC. DynEB identifies these applications and prefers a private L1 organization for them. Finally, for the insensitive non-shared-friendly applications (not shown due to lack of space), DynEB improves performance by 1%, 2%, and 4% over Best(Private,Shared++), Shared++, and private L1 baseline, respectively. This is because DynEB enables each core to adapt to the changing behavior of the executing application and obtain the advantages of both shared and private L1 organizations during different phases of execution.

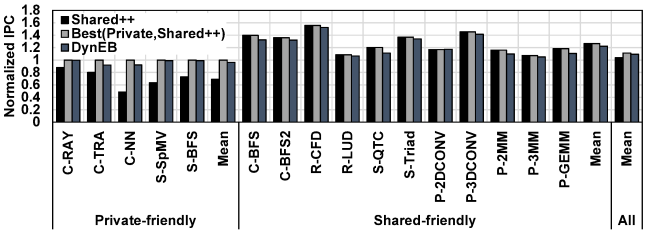


Figure 10: The effect of the proposed solutions on IPC. Results are normalized to the private L1 baseline.

Overall, DynEB improves performance of all evaluated applications by 9%. To demonstrate that, in Figure 11, we show normalized speedup for the evaluated applications sorted ascendingly. This is under the shared L1 organization (*Shared*), the optimizations in Section 3.3 (*Shared+Chunk* and *Shared++*), and the dynamic scheme (*DynEB*). We observe that although Shared+Chunk and Shared++ push the tail of the S-curve toward the private L1 organization, they still suffer due to the private-friendly applications. However, DynEB can recover the performance loss of these applications.

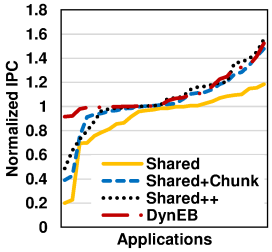


Figure 11: The effect of the proposed solutions on IPC (normalized to baseline) as S-curve.

Effect on L1 Miss Rate. Figure 13 shows how effective our solutions are for decreasing L1 miss rate. The results are normalized to the private L1 baseline. We observe the following. First,

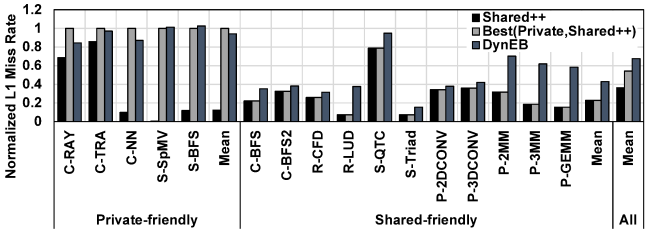


Figure 13: The effect of the proposed solutions on L1 miss rate. Results are normalized to the private L1 baseline.

Shared++ leads to lower L1 miss rates compared to the private L1 organization because of the extra effective capacity achieved using a shared L1 organization. Specifically, with Shared++, the L1 miss rate drops by 77% and 88% for shared-friendly and private-friendly applications, respectively. As for the insensitive non-shared-friendly applications (not shown due to lack of space), Shared++ reduces the L1 miss rate by only 13% as these applications possess low data replication (Figure 1). Second, for shared-friendly applications, DynEB decreases the L1 miss rate by 57% compared to a private L1 organization. This is because DynEB aims to adapt to the shared-friendly nature of these applications and executes them under a shared L1 organization. However, DynEB causes a 88% increase in the L1 miss rate compared to Shared++ because it runs half the cache sets as private during sampling. Additionally, some cores may end up running under a private L1 organization during some execution phases, which may lead to replication across cores, and thus less effective capacity and higher L1 miss rate. Specifically, Figure 12 quantifies the number of replicas across the cores under both private and shared L1 cache organizations and under DynEB. As expected, with Shared++, we maintain only a single copy of the data. However, under *Private*, each core can cache any data from the address range, which may lead to more replications across the cores (2.7 replicas on average). DynEB maintains fewer replicas compared to Private but more compared to Shared++ (1.4 replicas on average). This result conforms with the L1 miss rate increase under DynEB compared to Shared++. Finally, for the private-friendly applications, DynEB achieves an L1 miss rate similar to the private L1 baseline, as shown in Figure 13. These applications prefer a private L1 organization due to their high L1 hit rate and latency sensitivities, and DynEB runs them under their preferred organization.

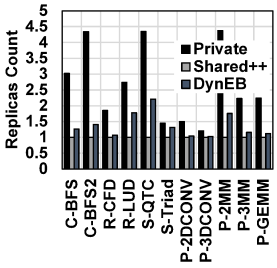


Figure 12: The effect of the proposed solutions on number of replicas.

Effect on Energy. The shared L1 organization introduces inter-core traffic. However, the chunking optimization (Section 3.3) reduces such overheads by only sending the data requested by the requester, not the entire line. Moreover, our proposed schemes reduce L2 and off-chip memory traffic. Using flit and hop counts as well as L2 and memory access counters, we use DSENT [61] and GPUWattch [39] to estimate energy consumption. Overall, the

total power under DynEB is similar to baseline, with <1% reduction averaged across all evaluated applications. Given the improvement in the overall throughput and execution time, the average energy savings under DynEB is 9% compared to the baseline. Therefore, DynEB improves performance-per-watt by 9% and the energy efficiency (performance-per-energy) by 20%, on average across all evaluated applications. For the shared-friendly applications, DynEB maintains the total power consumption (similar to baseline) and saves energy by 18%. Therefore, DynEB enhances performance-per-watt and energy efficiency for the shared-friendly applications by 22% and 49%, respectively.

Effect on Latency. Our private L1 baseline and proposed solutions assume a local L1 access latency of 28 cycles. The shared L1 cache organization imposes a latency overhead of 54 cycles, on average, for the communication between the requester and the home cores. Such inter-core communication overhead is insignificant compared to the 247 cycles, on average, to communicate with L2 in the baseline. Also, such latency overhead does not negatively affect the evaluated applications because of their latency-tolerant nature.

Adaptability of DynEB. The performance results so far show the versatility of DynEB. This is because DynEB utilizes a repeated two-step process of sampling and execution. Thus, DynEB adapts to the changing characteristics of a given application's execution. Also, DynEB is local per core. Hence, each core independently monitors application needs and decides the desirable mode of execution. To visualize this adaptive nature, Figure 14 shows how DynEB changes the execution mode under C-BFS and C-NN for a representative core. For both applications, DynEB identifies the desirable mode of execution and sticks to it for almost the entire execution.

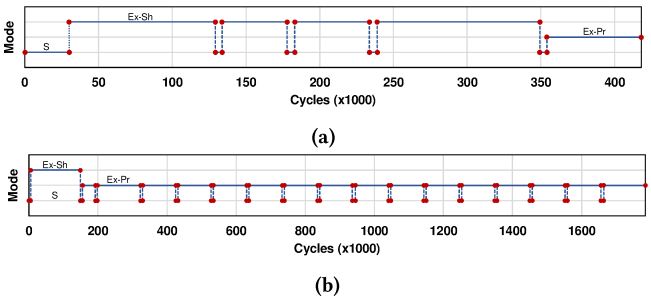


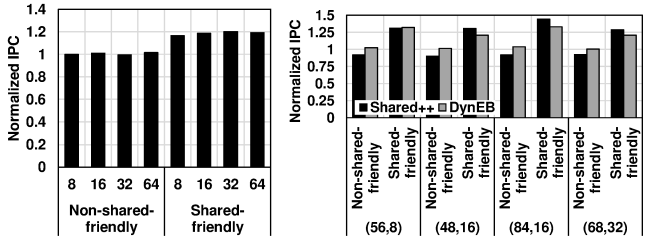
Figure 14: Execution timeline under DynEB for (a) C-BFS and (b) C-NN. *S* refers to a sampling phase. *Ex-Sh* and *Ex-Pr* refer to an execution under Shared++ and Private, respectively.

5.3 Sensitivity Studies

Effect of L1 Cache Size. We evaluate the effect of doubling the L1 cache size per core on the performance of our schemes. We observe that Shared++ and DynEB achieve around 11% improvement for the shared-friendly applications while maintaining the private performance of the non-shared-friendly applications, over a private baseline with 2× L1 cache size. The lower scope of the improvement is because the working set of some shared-friendly applications can now fit in the larger L1 cache. Additionally, some of these applications are latency-sensitive, making a shared L1 organization less desirable for them under 2× L1 cache size. We also compare Shared++ and DynEB, for the shared-friendly applications, under the baseline L1 cache size (Table 1) against a private L1 organization with double the L1 cache size, denoted as *Private(2x)*. We observe that Shared++ and DynEB improves IPC over Private(2x) by 8% and 4%, respectively. This shows that by enabling a shared L1 organization, we can perform better compared to a system with double the L1 cache resources without the extra cost/overhead of increasing the L1 cache size (84% cache area overhead).[6]

Effect of L2 Cache Size. We evaluate a boosted private L1 baseline with double the L2 cache size. We observe almost no performance improvement for the shared-friendly applications compared to the baseline. This is because performance is limited by the L2 reply bandwidth bottleneck [49, 73, 74]. Such a bottleneck is relieved with Shared++ and DynEB as the shared L1 organization utilizes the remote cores as an additional source of bandwidth.

Effect of L1 Access Latency. In our baseline and proposed schemes, we assume 28 cycles access latency for the L1 caches. Figure 15a shows average performance with DynEB under different L1 access latency, ranging from 8 to 64 cycles, each normalized to its respective private L1 baseline. We observe that DynEB achieves 17% performance improvement for the shared-friendly applications even under an L1 access latency of 8 cycles while maintaining the performance of the non-shared-friendly applications.



(a) L1 access latency **(b) Core and uncore components**

Figure 15: Sensitivity studies.

Effect of Core Count. We study the scalability of Shared++ and DynEB using 8×8 mesh and 10×10 mesh NoCs under two different configurations. Figure 15b shows performance of both Shared++ and DynEB normalized to their respective private L1 organization baseline. The notation in the figure is (number of cores, number of memory partitions). We observe that IPC follows a similar trend to what we observed using the baseline (28,8) 6 × 6 mesh. Specifically, with DynEB, we gain significant IPC improvement for the shared-friendly applications and maintain the private performance for the non-shared-friendly applications. For example, for an (84,16) system, DynEB improves IPC by 33% and 4%, on average, for the shared-friendly and non-shared-friendly applications, respectively. We observe higher IPC improvement under increased core count because the overall L1 capacity increases with more cores, thus the available collective L1 bandwidth increases under shared L1 organization. Also, with more cores, the *home camping* effect is reduced. Home camping, which is similar to partition camping [1], is caused by memory accesses that are skewed towards a subset of the home cores, which may degrade the performance. Thus, by increasing the core count, each core is assigned a smaller slice of

[6]The cache area overhead is estimated using CACTI 6.5 [44].

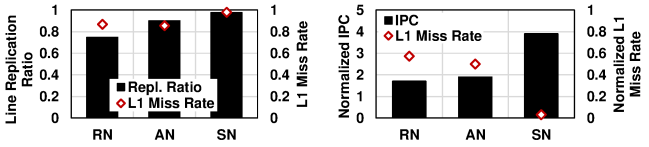
the address range which should likely lead to a uniform traffic distribution among the home cores and hence scales performance.

Effect of Additional Memory Partitions. Figure 15b shows the effect of increasing the memory partitions count (this increases total L2 capacity, L2 bandwidth, and memory bandwidth). For an 8×8 mesh, we study systems with 8 and 16 memory partitions. For a 10×10 mesh, we study systems with 16 and 32 memory partitions. We observe that for the systems with a smaller number of memory partitions, our schemes achieve performance boost at least as good as the systems with a greater number of memory partitions. This is because our schemes are more beneficial with more cores.

Effect of Core to Memory Partition Ratio. Figure 15b shows that our schemes can boost IPC for the shared-friendly applications under varying core-to-memory partition ratio. Even in a large (68,32) system, DynEB achieves 21% IPC improvement over the baseline (68,32) 10×10 mesh.

5.4 Case Study: Deep-Learning Applications

In this section, we briefly characterize three popular deep-learning workloads from Tango benchmark suite [28], namely AlexNet (AN), ResNet (RN), and SqueezeNet (SN). Additionally, we evaluate their performance under DynEB assuming a big 76-core system with 24 memory partitions (using 10×10 mesh) to mimic recent GPUs oriented to processing deep-learning applications. Figure 16a characterizes the evaluated applications in terms of L1 miss rate and cache line replication ratio (Section 2). We observe high replication ratio (up to 98% for SN) and high L1 miss rate (up to 98% for SN) in the evaluated applications, making them perfect candidates for our proposed schemes. Reducing this significant replication across the L1s enables more data to be cached on-chip, which boosts the L1 hit rate, on-chip bandwidth, and overall performance, as shown in Figure 16b. Specifically, on average, DynEB reduces the L1 miss rate by 79% for these applications, thus improving their performance by up to 3.9× and by 2.3× on average.



(a) Characterization (b) Performance

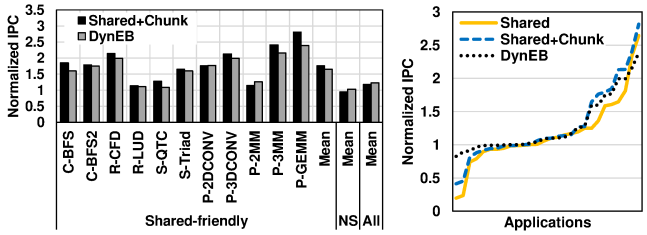
Figure 16: Analyzing deep-learning applications in terms of L1 miss rate, cache line replication ratio, and performance improvement under DynEB.

5.5 Case Study: Crossbar-based Shared L1 Cache Design

In this section, we evaluate the shared L1 organization under a crossbar NoC. A conventional crossbar connecting cores on one side of the crossbar to L2 slices on the other does not support inter-core communication. Therefore, in this case study, we investigate enabling such communication via the L2 slices. Then, we propose using work distribution crossbar [16, 18], which is already utilized in the graphics (rendering) pipeline, to forward inter-core traffic.

Inter-core Communication via L2 Slices. We update the L2 slices to simply receive a remote request/reply from a requester/home core and forward it back to the target home/requester core. We observe that using L2 to forward the inter-core traffic reduces performance by 23% compared to the private L1 organization. This is due to the contention between L2 replies and forwarded remote traffic, thereby significantly delaying the remote traffic and thus losing performance.

Inter-core Communication via Work Distribution Crossbar. We propose to utilize the work distribution crossbar [16, 18], which already exists and is used by the graphics pipeline, to handle inter-core traffic instead of using the L2 slices. The work distribution crossbar is a scalable multistage butterfly NoC that supports 1) the distribution of triangle and fragment work necessary for load balancing and 2) the synchronization communication necessary for ordering in the graphics pipeline [16]. Therefore, the work distribution crossbar inherently enables inter-core communication. A multistage butterfly (k-ary n-fly) supports a system with up to k^n nodes organized in n stages, where each stage has k^{n-1} switches with a radix k (i.e., $k \times k$ crossbar switch). For our 36-node baseline system (28 cores and 8 memory partitions), we assume a 6-ary 2-fly butterfly NoC. Figure 17a shows performance of *Shared+Chunk* (Section 3.3) and *DynEB* (Section 4.2) under the work distribution crossbar. We observe the following. First, Shared+Chunk and DynEB improve performance of the shared-friendly applications, on average, by 76% and 65%, respectively. Second, for the non-shared-friendly applications (denoted as *NS* in Figure 17a), Shared+Chunk incurs a 5% performance drop, on average. However, DynEB maintains these applications' private performance and offers a 2% performance improvement, on average. This is because DynEB obtains the advantages of both shared and private L1 organizations per each application needs.



(a) (b)

Figure 17: Performance of the shared L1 organization in terms of IPC under a crossbar-based system. *NS* refers to non-shared-friendly applications. Results are normalized to a crossbar-based system with private L1 organization.

Overall, Shared+Chunk and DynEB improve performance of all evaluated applications (denoted as *All* in Figure 17a) by 18% and 23%, respectively. To demonstrate that, Figure 17b summarizes the effect of the shared L1 organization (*Shared*), the proposed chunking optimization (*Shared+Chunk*), and the dynamic scheme (*DynEB*) on the evaluated applications sorted ascendingly. Similar to the mesh-based system, Shared and Shared+Chunk can provide performance benefits for the shared-friendly applications, but they fail to push the tail of the S-curve towards the private L1 baseline

due to the private-friendly applications. On the other hand, DynEB recovers the lost performance of the private-friendly applications, while improving the shared-friendly applications.

Scalability. We study the scalability of Shared+Chunk and DynEB for a 64-node system under two different configurations. Specifically, we evaluate a 48-core system with 16 memory partitions and a 56-core system with 8 memory partitions. For both configurations, we assume a 4-ary 3-fly butterfly NoC. Figure 18 shows performance of both Shared+Chunk and DynEB normalized to their respective private L1 baseline. The notation in the figure is (number of cores, number of memory partitions). We observe a similar trend to what we observed with the 36-node system. In particular, Shared+Chunk significantly boosts performance of shared-friendly applications while falling short for non-shared-friendly applications. On the other hand, DynEB matches the performance boost of the Shared+Chunk for shared-friendly applications and maintains the private performance for non-shared-friendly applications. Additionally, similar to our observation in Section 5.3, performance improvement under the 56-core system is higher compared to the 48-core system. This is because our proposed shared L1 organization benefits more in the presence of more L1 caches.

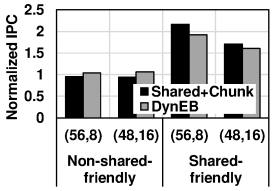


Figure 18: Crossbar-based system scalability.

6 RELATED WORK

To our knowledge, this is the first work to make a case for using shared L1 caches in GPUs. In this section, we briefly discuss works that are most relevant to this study.

Intra-core Locality in GPUs. There is a large body of work that focuses on exploiting the locality that exists within a private local L1 cache in GPUs [27, 29, 42, 54, 55, 58]. In this work, we specifically focus on the locality that exists across L1 caches. Multiple prior CTA schedulers [3, 38, 62] have used different heuristics to exploit the locality across CTAs. However, they are not ideal [26, 40, 66], and the fundamental problem of cache line replication across private L1 caches remains. While the goal of these schedulers is to improve cache performance, our approach 1) is not dependent on any scheduling algorithm, 2) does not require any software support to determine private and shared data, and 3) does not only reduce replication but can eliminate it. In general, prior L1 cache capacity management works based on bypassing [34, 62], sectoring [53], or compression [4] do not ensure zero data replication across L1s. However, they can continue to improve the performance of local L1 caches while our shared L1 organization can facilitate coordination across L1s for their better utilization.

Inter-core Locality in GPUs. Prior works proposed mechanisms to exploit inter-core locality in GPUs by allowing communication between multiple L1s via connecting L1s through a ring network [15], using the L2 cache to forward inter-core traffic [74], or coherence-like mechanisms [64]. Recently, Ibrahim *et al.* [23] aimed to further optimize inter-core communication using data sharing prediction and parallel probing/searching schemes. Although these works identified and exploited inter-core locality via inter-core communication, they do not provide a way to reduce or eliminate data replication across L1 caches as we do. Our shared L1 organization utilizes inter-core communication to eliminate the L1 cache wastage without the need for searching or prediction. Zhao *et al.*[73] boost performance of applications with high degrees of data sharing between cores by replicating the shared cache lines across different L2 slices. This is complementary to our work as ours improves the L1 bandwidth utilization while their work improves the L2 bandwidth.

Replication Control in CPUs. Many works have investigated the trade-offs between shared and private caches in the context of CPUs. These works use a flavor of replication control [7, 11, 21, 35, 43, 65, 71], cooperative capacity management mechanisms across cores [9, 14, 20, 36, 51, 56], hybridized shared/private designs [37, 72], OS-level techniques [12, 19], or focus on different architectures/components [8, 59]. Our work differs from those in multiple aspects. First, most of the replication management works in the CPU context consider latency as an important metric for controlling replication. We show that using a latency metric (i.e., AMAT) performs poorly in GPUs as it does not consider the latency-tolerance property of applications. Therefore, we investigate a GPU-oriented metric (i.e., EB) to gauge an application's affinity towards a private or shared L1 organization. Second, all works in the CPU context investigate the aforementioned approaches for the last-level caches as L1 caches always aim to reduce latency. Due to the latency-tolerant and throughput-oriented behavior of GPUs, optimizing for hit rate (and hence bandwidth) is usually more important than optimizing for latency, so we consider using a shared cache organization for L1 caches. Finally, our mechanism is entirely locally managed, and no coordinated mechanisms are needed to make a decision.

7 CONCLUSIONS

In this work, we show that using a shared L1 cache organization in GPUs is attractive in terms of performance for many applications. We also address the challenges related to applications that lose performance from such an organization with low-overhead communication optimization techniques and a lightweight dynamic mechanism that gauges an application's affinity towards a private or shared L1 organization and configures the L1 caches accordingly. We show that our techniques can boost performance and can be even more beneficial for future large GPUs with many cores. We hope that this work will open up new research directions in sharing other resources in the GPU (e.g., software-managed caches).

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers, Nuwan Jayasena, and members of the Insight Computer Architecture Lab at William & Mary for their feedback. This material is based upon work supported by the National Science Foundation (NSF) CAREER award (#1750667). This work was performed in part using computing facilities at William & Mary.

REFERENCES

[1] Ashwin M. Aji, Mayank Daga, and Wu-chun Feng. 2011. Bounding the Effect of Partition Camping in GPU Kernels. In *Proceedings of the International Conference on Computing Frontiers (CF)*.

[2] AMD. 2019. AMD RDNA Architecture White Paper. (August 2019). https://www.amd.com/system/files/documents/rdna-whitepaper.pdf

[3] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[4] Akhil Arunkumar, Shin-Ying Lee, Vignesh Soundararajan, and Carole-Jean Wu. 2018. LATTE-CC: Latency Tolerance Aware Adaptive Cache Compression Management for Energy Efficient GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[5] Ali Bakhoda, John Kim, and Tor M Aamodt. 2010. Throughput-Effective On-Chip Networks for Manycore Accelerators. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[6] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[7] Bradford M Beckmann, Michael R Marty, and David A Wood. 2006. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[8] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. 2018. Scalable Distributed Last-level TLBs Using Low-latency Interconnects. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[9] Jichuan Chang and Gurindar S. Sohi. 2006. Cooperative Caching for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[10] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.

[11] Zeshan Chishti, Michael D Powell, and TN Vijaykumar. 2005. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[12] Sangyeun Cho and Lei Jin. 2006. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[13] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In *Proceedings of the Workshop on General Purpose Processing Using GPU (GPGPU)*.

[14] R. G. Dreslinski, D. Fick, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, Nurrachman Liu, M. Wieckowski, Gregory Chen, T. Mudge, D. Sylvester, and D. Blaauw. 2012. Centip3De: A 64-Core, 3D Stacked, Near-Threshold System. In *Proceedings of the Symposium on High Performance Chips (Hot Chips)*.

[15] Saumay Dublish, Vijay Nagarajan, and Nigel Topham. 2016. Cooperative Caching for GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* (2016).

[16] Matthew Eldridge, Homan Igehy, and Pat Hanrahan. 2000. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*.

[17] GPGPU-Sim v3.x. 2017. Address Mapping. (June 2017). http://gpgpu-sim.org/manual/index.php5/GPGPU-Sim_3.x_Manual#Memory_Partition

[18] Ayub A. Gubran and Tor M. Aamodt. 2019. Emerald: Graphics Modeling for SoC Systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[19] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[20] H. Hossain, S. Dwarkadas, and M. C. Huang. 2008. Improving Support for Locality and Fine-grain Sharing in Chip Multiprocessors. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

[21] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. 2007. A NUCA Substrate for Flexible CMP Cache Sharing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2007).

[22] Hynix. 2009. Hynix GDDR5 SGRAM Part H5GQ1H24AFR Revision 1.0. (2009). http://www.hynix.com/datasheet/pdf/graphics/H5GQ1H24AFR(Rev1.0).pdf

[23] Mohamed Assem Ibrahim, Hongyuan Liu, Onur Kayiran, and Adwait Jog. 2019. Analyzing and Leveraging Remote-core Bandwidth for Enhanced Performance in GPUs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

[24] W. Jia, K. A. Shaw, and M. Martonosi. 2014. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[25] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Nilardish Chatterjee, Steve Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*.

[26] Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. Orchestrated Scheduling and Prefetching for GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[27] Adwait Jog, Onur Kayiran, Nachiappan C. Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. 2013. OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[28] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Detailed Characterization of Deep Neural Networks on GPUs and FPGAs. In *Proceedings of the Workshop on General Purpose Processing Using GPU (GPGPU)*.

[29] Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. 2013. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

[30] Onur Kayiran, Nachiappan Chidambaram Nachiappan, Adwait Jog, Rachata Ausavarungnirun, Mahmut T. Kandemir, Gabriel H. Loh, Onur Mutlu, and Chita R. Das. 2014. Managing GPU Concurrency in Heterogeneous Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[31] Mahmoud Khairy, Akshay Jain, Tor M. Aamodt, and Timothy G. Rogers. 2018. Exploring Modern GPU Memory System Design Challenges through Accurate Modeling. *arXiv* (October 2018).

[32] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. 2015. Stash: Have Your Scratchpad and Cache It Too. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[33] G. Koo, H. Jeon, and M. Annavaram. 2015. Revealing Critical Loads and Hidden Data Locality in GPGPU Applications. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*.

[34] Gunjae Koo, Yunho Oh, Won Woo Ro, and Murali Annavaram. 2017. Access Pattern-Aware Cache Management for Improving Data Utilization in GPU. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[35] George Kurian, Srinivas Devadas, and Omer Khan. 2014. Locality-Aware Data Replication in the Last-Level Cache. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[36] George Kurian, Omer Khan, and Srinivas Devadas. 2013. The Locality-aware Adaptive Cache Coherence Protocol. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[37] Woo-Cheol Kwon, Tushar Krishna, and Li-Shiuan Peh. 2014. Locality-oblivious Cache Organization Leveraging Single-cycle Multi-hop NoCs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[38] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu. 2014. Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[39] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[40] Ang Li, Shuaiwen Leon Song, Weifeng Liu, Xu Liu, Akash Kumar, and Henk Corporaal. 2017. Locality-Aware CTA Clustering for Modern GPUs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[41] D. Li and T. M. Aamodt. 2016. Inter-Core Locality Aware Memory Scheduling. *IEEE Computer Architecture Letters (CAL)* (2016).

[42] Dong Li, Minsoo Rhu, Daniel R Johnson, O Mike, Mattan Erez, Doug Burger, Donald S Fussell, and Stephen W Redder. 2015. Priority-Based Cache Allocation in Throughput Processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[43] Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. 2004. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[44] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[45] C. Nugteren, G. van den Braak, H. Corporaal, and H. Bal. 2014. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[46] NVIDIA. 2011. CUDA C/C++ SDK Code Samples. (2011). http://developer.nvidia.com/cuda-cc-sdk-code-samples

[47] NVIDIA. 2019. CUDA C++ Programming Guide. (November 2019). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[48] NVIDIA. 2019. Parallel Thread Execution ISA Version 6.5. (November 2019). https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[49] Ashutosh Pattnaik, Xulong Tang, Onur Kayiran, Adwait Jog, Asit Mishra, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. 2019. Opportunistic Computing in GPU Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[50] Louis-Noël Pouchet. 2012. Polybench: The Polyhedral Benchmark Suite. (2012). http://web.cs.ucla.edu/~pouchet/software/polybench/

[51] Moinuddin K Qureshi. 2009. Adaptive Spill-Receive for Robust High-Performance Caching in CMPs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[52] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans. 2020. HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[53] Minsoo Rhu, Michael Sullivan, Jingwen Leng, and Mattan Erez. 2013. A Locality-aware Memory Hierarchy for Energy-efficient GPU Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[54] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[55] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2013. Divergence-Aware Warp Scheduling. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[56] Dyer Rolan, Basilio B Fraguela, and Ramon Doallo. 2012. Adaptive Set-Granular Cooperative Caching. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[57] D. Sanchez and C. Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[58] Ankit Sethia and Scott Mahlke. 2014. Equalizer: Dynamic Tuning of GPU Resources for Efficient Execution. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[59] Amna Shahab, Mingcan Zhu, Artemiy Margaritov, and Boris Grot. 2018. Farewell My Shared LLC!: A Case for Private Die-stacked DRAM Caches for Servers. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[60] Inderpreet Singh, Arrvindh Shriraman, Wilson WL Fung, Mike O'Connor, and Tor M Aamodt. 2013. Cache Coherence for GPU Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[61] C. Sun, C. H. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. S. Peh, and V. Stojanovic. 2012. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *Proceedings of the International Symposium on Networks-on-Chip (NOCS)*.

[62] Abdulaziz Tabbakh, Murali Annavaram, and Xuehai Qian. 2017. Power Efficient Sharing-Aware GPU Data Management. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*.

[63] A. Tabbakh, X. Qian, and M. Annavaram. 2018. G-TSC: Timestamp Based Coherence for GPUs. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[64] David Tarjan and Kevin Skadron. 2010. The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[65] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Nexus: A New Approach to Replication in Distributed Shared Caches. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

[66] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu. 2018. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[67] Haonan Wang, Fan Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. 2018. Efficient and Fair Multi-programming in GPUs via Effective Bandwidth Management. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*.

[68] Wm A Wulf and Sally A McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News* (1995).

[69] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa. 2019. Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[70] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie. 2016. OSCAR: Orchestrating STT-RAM Cache Traffic for Heterogeneous CPU-GPU Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*.

[71] Michael Zhang and Krste Asanovic. 2005. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[72] Li Zhao, Ravi Iyer, Mike Upton, and Don Newell. 2008. Towards Hybrid Last Level Caches for Chip-Multiprocessors. *ACM SIGARCH Computer Architecture News* (2008).

[73] Xia Zhao, Almutaz Adileh, Zhibin Yu, Zhiying Wang, Aamer Jaleel, and Lieven Eeckhout. 2019. Adaptive Memory-Side Last-Level GPU Caching. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[74] X. Zhao, Y. Liu, A. Adileh, and L. Eeckhout. 2017. LA-LLC: Inter-Core Locality-Aware Last-Level Cache to Exploit Many-to-Many Traffic in GPGPUs. *IEEE Computer Architecture Letters (CAL)* (2017).